Large Limits to Software Estimation

J. P. Lewis
Disney TSL
3100 Thornton Ave.,
Burbank CA 91506 USA
zilla@computer.org

Abstract

Algorithmic (KCS) complexity results can be interpreted as indicating some limits to software estimation. While these limits are abstract they nevertheless contradict enthusiastic claims occasionally made by commercial software estimation advocates. Specifically, if it is accepted that algorithmic complexity is an appropriate definition of the complexity of a programming project, then claims of purely objective estimation of project complexity, development time, and programmer productivity are necessarily incorrect.

Keywords
Estimation and metrics, project management, risks, ethical issues.

Introduction

Among the important practical problems in software engineering is *software estimation* — the estimation of development schedules and the assessment of productivity and quality. Is it possible to apply mathematical and scientific principles to software estimation, so that development schedules, productivity, and quality might be objectively ascertained or estimated rather than being a matter of opinion?

The debate over this question is familiar. In the case of development schedules, for example, many programmers find it self evident that accurate and objective estimates are not possible. One reader of an early version of this paper commented, "Software practitioners know about poor predictability from empirical evidence. I don't need to prove it..."

On the other hand, there are a large number of design methods, development processes, and programming methodologies that claim or hint at objective estimation of development schedules, project complexity, and programmer productivity. For example, a handbook of software quality assurance describes the benefits of a quality management process [16]:

> "In the Certainty state [of quality management], the objective of software development and software quality management, producing quality software on time with a set cost everytime, is possible."

A software process manifesto states [14]

> "[In a mature organization] There is an objective quantitative basis for judging product quality and analyzing problems with the product and process. Schedules and budgets are based on historical performance and are realistic"

Similarly, a book promoting a software estimation package [15] states that "...software estimating can be a *science*, not just an *art*. It really is possible to accurately and consistently estimate costs and schedules for a wide range of projects," etc.

The answer to our question ('can software be objectively estimated?') has both practical and ethical implications. Newspaper headlines frequently describe the cancellation of costly software projects that are behind schedule and over budget. With computer programs now widely deployed in socially critical roles it is recognized that software professionals have a responsibility to make accurate and truthful characterizations of prospective software sytems.

Given the existence of the various software methodologies and processes alluded to above, it would be easy to conclude that the problem is merely that these methods are not being practiced. On the other hand, considering the wide variety of competing methodologies and the well considered critiques of some of these methodologies [1, 2, 3, 4, 8], one may be tempted to adopt an outside perspective and ask whether all of the stated goals of these methodologies are possible even in principle.

In this paper we will look at software estimation from the point of view of algorithmic or KCS (Kolmogorov-Chaitin-Solomonoff) complexity. Section two introduces the notion of algorithmic complexity. In sections three and four we will find that algorithmic complexity results can be directly interpreted as indicating that software complexity, development schedules, and productivity cannot be objectively and feasibly estimated and so will remain a matter of opinion. Section five considers approximate and statistical estimators. The situation here is more optimistic, but we give reasons for maintaining a scepticism towards overly optimistic claims of estimation accuracy. Although these are negative results, we will argue in concluding that the software industry can benefit from acknowledging and publicizing the inherent limitations and risks of software development.

ALGORITHMIC COMPLEXITY

Algorithmic complexity (AC) defines the complexity of a digital object to be the length of the shortest computer program that produces that object. This definition formalizes an intuitive notion of complexity. Consider the three patterns:

```
11111111111111...
12312312312312...
30547430729732...
```

These strings may be of the same length but the first two strings appear to be simpler than the third. This subjective ranking is reflected in the length of the programs needed to produce these strings. For the first string the program is a few bytes in length,

```
  for i:=1 to n print('1');
```

The program for the second string is slightly longer since it will contain either nested loops or the literal `'123'`. If there is no obvious pattern to the third string, the shortest program to produce it is the program that includes the whole string as literal data and prints it — the string is incompressible or algorithmically random.

Ideally the complexity of an object should be a property only of the object itself, but the choice of computer and programming language affects program lengths. Algorithmic complexity handles this issue by considering the complexity to be well defined only for large objects. The choice of an inelegant language or machine adds only a constant amount to the algorithmic complexity, since a translator or simulator from any language or machine to any other is a fixed-size program. In the limit of large objects this fixed size becomes insignificant.

There are several major variants of AC (endmarker complexity, prefix complexity, Chaitin's prefix variant in which conditional complexity is conditioned on a shortest program rather than on an uncompressed string), but they are asymptotically similar. The AC of a string will be denoted $AC(x)$ when our purpose does not distinguish these variants; we will switch to the notation $K(x)$ (prefix complexity) for the discussion of approximate bounds.

The flavor of algorithmic complexity reasoning will be shown with the following theorem (it will also be used later in the paper):

Chaitin Incompleteness theorem. *A formal theory with N bits of axioms cannot prove statements of the form '$AC(x) > c$' if c is much greater than N.*

The proof is by contradiction. One makes the reasonable assumption that if a statement $AC(x) > c$ can be proved then it should be possible to extract from the proof the particular x that is used. Then by appending this extraction to the proof sequence (of $AC \approx N$) one can generate the string x using approximately N bits. But the proof has shown that the AC of x is $AC(x) > c > N$ resulting in contradiction.

The proof is illuminated by recasting the formal system in computational form. A formal system consists of a set of symbols; a grammar for combining the symbols into statements; a set of axioms, or statements that are accepted without proof; and rules of inference for deriving new statements (theorems). A proof is a listing of the sequence of inferences that derive a theorem. It is required that a proof be formally (i.e. mechanically) verifiable. Thus, there is a correspondence [18] between a formal system and a computational system whereby a proof is essentially a string processing computation:[1]

[1]This is an arguably idealized and restrictive notion of proof; see Naur [12] for a discussion. Also note that there are alternate ways of defining the correspondence between formal and computational systems.

axioms	$\Longleftrightarrow$	program input or initial state
rules of inference	$\Longleftrightarrow$	program interpreter
theorem(s)	$\Longleftrightarrow$	program output
derivation	$\Longleftrightarrow$	computation

Consider a program that will successively enumerate all possible proofs in length order until it finds a proof of the form $AC(x) > c$, which it will then output. The program and computer will need to encode the axioms and rules of inference, which are N bits by assumption. The program will also need to encode the constant c. Say that the size of the search program including the encoded c is $AC(c) + N$. Append a second algorithmically simple program that extracts and prints the x from the proof found in the search. Let k be the AC of the second program, plus, if necessary, the AC of some scheme for delimiting and sequentially running the two programs. We now have a compound program of AC $AC(c) + N + k$.

It is possible to choose a c, such as $2^{2^{2^{\cdots}}}$, that is much larger than N but whose AC is small. If c is picked as $c > AC(c)+N+k$ then either we have a program of $AC < c$ that generates a string that is proved to have $AC > c$ (such a proof is then wrong, so the formal system is unsound) or the program cannot prove statements of the form $AC(x) > c$. Since the details of the formal system were not specified, it is concluded that no formal system can prove that strings are much more complicated than the axioms of the system itself.

Definitions
A *formal process* is a specified and repeatable process that can be followed by independent agents (computer or human) to arrive at a common conclusion. We assume the Church-Turing thesis, that any such process is essentially an algorithm even if the process is in fact followed by humans rather than a computer.

An *objective* estimate is an estimate obtained via a formal process.

A *feasible* process is one that can be completed with conceivably realizable time and other resources. Exponential time algorithms, such as searching over the space of programs for one satisfying some objective, are clearly infeasible. For example, searching all possible program texts up to 100 bytes in length would require considering some fraction of $O(2^{800})$ possible texts – assuming a processor capable of examining a billion (10^9) texts per second this task would still require a number of centuries (far) larger than can easily be described by common words denoting large numbers ('billion', etc.).

Additional background on computability and KCS complexity is found in textbooks [17, 10].

COMPLEXITY AND SCHEDULE ESTIMATES

Various software design methods and processes address the issue of predicting development times. Software *cost models* estimate development time as a function of a size measure such as source line counts or function points. Software process literature and commercial software management tools have suggested that cost models can be combined with historical data on development times to predict the development times of future projects.

In an extensive empirical study Kemerer benchmarked four software cost estimation algorithms on data gathered from 15 large projects for which accurate records were available. It was found that these models had only limited predictive ability in ex post facto estimation of the development times for completed projects — the selected error measure (magnitude of error normalized by the actual development time) ranged from 85 percent to more than 700 percent [9]. Kemerer indicates that the limited accuracy of these models may be accounted for by variations in problem domain and other factors, and suggests that the models may be tuned to be more accurate.

The limited accuracy of these cost models is not the fundamental obstacle to software estimation however. Rather, since cost models are a function of a size or complexity measure, the issue is how to estimate the size or complexity of a new project.

Consider the following scenario: A software company wishes to improve the predictability of its software process, so it decides to gather statistical data on development times. As part of this effort each programmer is assigned a timed series of exercises. It is found that the average programmer at the company can complete each exercise in an average of 3.7 hours. Now the company is asked to bid on the development of an operating system for a major company that is years behind schedule on their own operating system development project. Based on the newly gathered statistics, the company estimates that it can deliver the new operating system in about 3.7 hours using one average programmer.

This absurd example is intended to clearly illustrate that estimates of development time depend on estimates of the size or complexity of a new program, and *historical statistics cannot provide the latter.* In the preceding example the complexity of the proposed operating system is presumably much greater than the complexity of the programming exercises, but the data do not say anything about the relative difference in complexity. Can complexity itself be formally and feasibly determined or estimated a priori?

Claim 1: Program size and complexity cannot be feasibly estimated a priori.

Algorithmic complexity shows that the minimal program size for a desired task cannot be feasibly computed, and a trivial upper bound on program size exists but is not useful.

Before discussing these results further we need to relate algorithmic complexity to real-world programs. Recall that algorithmic complexity is defined as the minimum program size needed to produce a desired output string. The complexity of a program that produces a fixed output will be defined as the AC of that output. Since this definition deals with output only we will briefly indicate how arguments, state, and interactivity might be accommodated:

- *interactivity:* An interactive program can be considered as a collection of subprograms that are called in sequence according to user commands. These subprograms will share subroutines and a global state. Ignoring arguments and state for the moment, the AC of the program is the AC of the combined subprograms, plus the AC of a small event loop that calls the subprograms based on user input.[2]
- *arguments:* The AC of a function that depends on arguments can be defined as the AC of a large table containing the argument-value pairs interleaved in some fashion, plus the AC of some scheme for delimiting the argument-value pairs, plus the AC of a small program that retrieves an output given the corresponding input. The size of this uncompressed tabular representation will be called *tabular size*.
- *state:* State can be considered as an implicit argument to any routines whose behavior is affected.

These comments are only a rough sketch at formulating the AC of real-world programs, but the fidelity of this formulation is not crucial to our argument: if the complexity of an output-only program cannot be objectively determined, the addition of arguments, state, and interactivity will not simplify things.

The following central results of algorithmic complexity show that complexity is not feasibly computable.

- *KCS noncomputability theorem: there is no algorithm for computing the AC of an arbitrary string.* Denote a shortest program for producing a particular object x as x^*: $AC(x^*|x)$ is not recursive (computable). Rephrasing this for our purposes, there is no algorithm for finding the shortest program with a desired behavior.
- *A trivial upper bound on program size (tabular size) can be defined but is not feasible.* A trivial upper bound on program size is easy to define — it is simply size of the output string or argument table describing the program's behavior, as sketched above. This 'tabular size' bound is not feasible however. Consider a simple function that accepts two 32-bit integer arguments and produces an integer result. This function can be represented as an integer-valued table with 2^{64} entries. While small data types such as characters are sometimes processed in a tabular fashion, this example makes it clear that tabular specification becomes infeasible even for small functions involving several integers.
- *An asymptotic upper bound to AC can be computed, but not feasibly.* One can write a search program that enumerates all programs smaller than the tabular size in lexicographic order, looking for the shortest one with the desired behavior. Since many programs will loop the search program must interleave the enumeration and execution of the programs. That is, it runs each program constructed so far for a certain number of time steps, collects results on programs that finish during this time, and then it constructs the next program and

[2]It has recently been argued that interactive programs cannot be considered as conventional Turing machine-equivalent algorithms, basically because (according to this argument) human interaction should be considered as an intrinsic part of an interactive program. In any particular run of a program, however, the human input can be replaced with a recording of that input with no changes to either the behavior of the program or to the program text. The AC of a program is therefore independent of this debate over whether interactive programs should be considered as Turing machine-equivalent.

begins running it along with previously constructed programs that are still active. The search program will asymptotically identify smaller programs with the desired behavior but there is no way to know if a long-running program is looping or if it will finish and prove to be a still smaller representation of the desired function. This approach is infeasible, in part because the number of programs to be checked is exponential in the tabular size.

The preceding comments indicate that there is no way to objectively define the algorithmic complexity of a program. The minimal program size cannot be feasibly computed, and the trivial upper bound vastly overestimates the size of a realistic program. In fact, the ratio between the tabular size and the (uncomputable) AC, considered as a function of the tabular size, is known to grow as fast as any computable function.

Claim 2: Development time cannot be objectively predicted.

Since it is clear that development time estimates must consider program size or complexity among other factors, this claim is a direct consequence of the fact that program size cannot be objectively predicted. There is some limit on the speed at which a programmer can write code, so any development time estimate that is supposed to be independent of program size will be wrong if the program turns out to be larger than can be constructed during the estimated time period.

These comments apply to programming problems in general. Is it possible to do better in particular cases? For example, suppose a company has developed a particular program. If it is asked to write the same program again, it now has an objective estimate of the program size and development time. In practice this is a common scenario, since it is often necessary to recode legacy software for a different language or platform. But is there a middle ground between having no objective estimate and an estimate based on an identical completed project? Clearly the parts of a project that are similar to previous projects will be estimated more accurately. The remaining parts, even if they are small, can be problematic. We can guess how long the coding will take, but since the necessary coding time for a even small function may range from a few hours to perhaps years (if the routine is equivalent to an as yet unsolved mathematical problem, c.f. the formal system↔computation equivalence described in section two), there is no way to objectively know in advance how long the development will take. Most experienced programmers have encountered projects where an apparently trivial subproblem turns out to be more difficult than the major anticipated problems.

ESTIMATION OF PRODUCTIVITY

A wide variety of programming disciplines and processes have been proposed in the past several decades. Many of these proposals are justified by way of claims that programming productivity is increased. On the other hand, recent studies have shown that programmers average only a few delivered lines of code per day. Have structured programming, object-oriented programming, CASE, 4GLs, object-oriented design, design patterns, or other trends resulted in greater productivity, and if so, is there any way of objectively determining which of these techniques result in the greatest productivity gains?

Claim 3: Absolute productivity cannot be objectively determined.

Consider a statement that N lines of source code were developed in T time, with N/T higher than measured on other projects. This suggests that higher productivity has been achieved. But productivity is relevantly defined as the speed of *solving the problem,* not the speed of developing lines of code. If the N is significantly higher than it needs to be for the particular problem then a high N/T ratio may actually represent low productivity. This is not merely a theoretical possibility: DeMarco and Lister's programming benchmarks empirically showed a 10-to-1 size range among programs written in the same language to the same specification [7].

We conclude that since there is no feasible way to determine program complexity, productivity cannot be compared across projects. This position has been arrived at previously using less formal arguments — it is commonly noted that measures such as lines of code and function points may not reflect domain and problem specific variations in complexity.

Proviso: The relative effectiveness of various software engineering methods can be determined by way of a comparative experiment in which a *fixed* problem is solved by programming teams using different methods. Since it is believed that that the effects of programmer variability are much stronger than those due to development methods (10-to-1 differences in productivity across programmers working on the same problem have been found [11]) a large experiment might be necessary to achieve valid results.

WHAT ABOUT APPROXIMATE ESTIMATORS?

Though we have argued that absolute algorithmic complexity cannot be estimated, there remains the possibility of a approximate or statistical estimator E, say of the form

$$AC(x) \leq E(x) \leq AC(x) + b$$

for some bound b; for practical purposes this would be quite useful. The situation in regards to such an estimator is not so clear cut, but the following discussion may suggest maintaining a scepticism towards claims of strong estimation accuracy, even of an approximate sort.

The Chaitin incompleteness theorem (section two) is relevant. Rephrased, it says that an approximate estimator program cannot produce a lower bound on the complexity of programs much larger than its own size.

We will now show that an approximate estimator E of the form indicated above also cannot bound complexity to within a fixed range. (We switch to the notation $K(x)$ indicating the prefix complexity.) Consider the supposed estimator as identifying a set B of programs of complexity $K(p) \ldots K(p)+b$ that contains the given program p. The complexity of the program can now be expressed using a two-part description, the first part that of identifying the set B; the second part that of identifying the particular program given B. Apply the triangle inequality

$$K(a|b) \leq K(a|x) + K(x|b) + O(1)$$

to the two-part description:

$$K(K(p)|p) \leq K(K(p)|B) + K(B|p) + O(1)$$

$K(K(p)|B) \leq \log_2 |B| + O(1)$: given the set, the size of a program to identify the given member is at most the size of a program that indexes into the set. Because $K(K(p)|p) \neq O(1)$ (complexity of complexity is not computable) this means that $K(B|p) \neq O(1)$, i.e.,
Claim 4: There is no estimator which produces a correct fixed bound on the complexity of all inputs.

There are weaker alternatives which are still open to consideration, e.g. a suite of programs each of which is only required to bound the complexity of some subset of inputs, etc. Also the $O(1)$ constants are unknown and so it is possible that an estimator could accurately bound the complexity of inputs up to some useful threshold complexity. The preceding discussion does suggest, however, that any claim of producing an accurate approximation to AC should be examined carefully.

More generally, statistical estimation of complexity has an intrinsic problem that does not arise with common applications of statistics such as estimating a population mean. The problem is that the ground truth is unknown and unknowable, so it is not possible to determine the bias and variance of different estimators. As such, independent and disagreeing observers are not immediately lead to a common conclusion because they can adopt differing estimators to support their opinions.

This issue resembles the problem of defining and measuring psychological characteristics such as intelligence. While there is no absolute and objective measure of intelligence, ‘intelligence’ can be somewhat circularly defined as the ability measured by an intelligence test. The merits of particular intelligence tests are then debated by considering their correlation with measurable things (such as school grades) that are considered to be related to intelligence. Similarly, there is no feasible objective measure of complexity, but ‘complexity’ can be defined as the property measured by a proposed code metric. The particular estimate must then be justified; this can be done by demonstrating a correlation between the estimate and an expected correlate of complexity such as measured development time. We conclude that approximate estimators should be selected based on empirical demonstrations of utility.

CONCLUSIONS

We have argued that program complexity (and hence productivity) cannot be objectively identified. Our conclusions will seem obvious to many, and have been arrived at previously using informal arguments. The algorithmic complexity perspective formalizes, strengthens and simplifies these arguments.

Revisiting our assumptions.

Our arguments rest on the consideration of algorithmic complexity as an appropriate definition of the complexity of programs. AC is a precise and developed notion of complexity and it has been applied in a variety of fields. The application of AC to the complexity of programs seems evident. For example, AC addresses the well known problems of code metrics – that they may not reflect the actual complexity of the code, for reasons such as coding style, choice of language, or other reasons. The selection of an appropriate definition of complexity is ultimately a point of philosophy, however, since it is establishing a correspondence between an intuitive notion and a formal one. We do not claim that AC is the only complexity suitable for classifying programs, but it is intuitively appealing and supports precise reasoning about related issues.

Some of the implications of our perspective on the software engineering debate will now be discussed.

Should software estimation be called “engineering”?

The answer to this question has important implications. If software estimation is believed to be a codifiable engineering process analogous to house building then litigation is a reasonable and expected consequence of inaccurate estimations. This and similar issues currently divide the software engineering community into two camps — a “process” camp, who believe that quality software can be developed on time if a particular software process or programming technology is used, and a “problem solving” camp, who believe that programming is fundamentally a process of solving problems and as such intrinsically resists codification. The problem solving viewpoint is represented in the software engineering literature by Bollinger [5], who writes

> “The creation of genuinely new software has far more in common with developing a new theory of physics than it does with producing cars or watches on an assembly line.”

Bollinger further argues that the process viewpoint is not just incorrect but possibly dangerous, since it focuses attention on codified procedures and away from the unknown and potentially risky issues in development. Our conclusion supports the problem solving viewpoint at least in so far as the opposing (process) viewpoint rests on hopes of objective estimation of software complexity.

Ethics

Though our conclusions may be considered a ‘negative result’, we agree with authors [6] who warn that exaggerated claims and overly optimistic estimates are harming the credibility of the software industry and inviting litigation and possible regulation. Credibility will not be achieved by continuing to promise that software predictability is just around the corner. Instead, the software industry should attend to the intrinsic uncertainties and risks of software development and where necessary promote the public discussion and honest assessment of these risks.

REFERENCES

[1] D. E. Avison, H. U. Shah, and D. N. Wilson, “Software Quality Standards in Practice: The Limitations of Using ISO-9001 to Support Software Development,” *Software Quality Journal* 3, p. 105-111, 1994.

[2] J. Bach, “The Immaturity of the CMM,” *American Programmer,* Sept. 1994.

[3] J. Bach, “Enough about Process: What We Need are Heros,” *IEEE Software* Vol. 12, No. 2, Feb. 1995, pp. 96-98.

[4] T. Bollinger and C. McGowan, “A Critical Look at Software Capability Evaluation,” *IEEE Software* Vol. 8, No. 4, pp. 25-41, 1991.

[5] T. Bollinger, “The Interplay of Art and Science in Software,” *IEEE Computer,* Oct. 1997, pp. 128, 125-126.

[6] R. Charette, “Are We Developers Liars or just Fools,” *IEEE Computer*, July 1995 pp. 90-92.

[7] T. DeMarco, *Why Does Software Cost So Much? and other Puzzles of the Information Age,* Dorset, New York, 1995.

[8] M. Fayad and M. Laitinen, “Process Assessment Considered Wasteful,” *Communications ACM*, Vol. 40, No. 11, November 1997, pp. 125-128.

[9] C. F. Kemerer, “An Empirical Validation of Software Cost Estimation Models,” *Communications ACM*, Vol. 30, No. 5, May 1987, pp. 416-429.

[10] M. Li and P. Vitányi, *An Introduction to Kolmogorov Complexity and its Applications*, 2nd Ed., Springer, New York, 1997.

[11] S. McConnell, *Rapid Development: Taming Wild Software Schedules*, Microsoft, Redmond, WA, 1996, p. 167.

[12] P. Naur, *Knowing and the Mystique of Logic and Rules,* Kluwer Academic, Dordrecht, 1995.

[13] P. Neumann and contributors, “Risks to the Public” column, *Software Engineering Notes*; also P. G. Neumann, B. Simons, and others, “Inside Risks” column, *Communications ACM.*

[14] M. C. Paulk, B. Curtis, M. B. Chrissis, C. V. Weber, *The Capability Maturity Model for Software* Version 1.1, CMU/SEI-93-TR-24 Feb. 93, p. 19.

[15] W. H. Roetzheim and R. A. Beasley, *Software Project Cost and Schedule Estimating: Best Practices*, Prentice Hall, Upper Saddle River, New Jersey, 1995, p. xvii.

[16] G. G. Schulmeyer, “Software Quality Lessons from the Quality Experts,” in G. G. Schulmeyer and J. I. McManus, Eds., *Handbook of Software Quality Assurance* (2nd ed.), Nelson Canada, 1995. p. 76.

[17] C. H. Smith, *A Recursive Introduction to the Theory of Computation*, Springer Verlag, New York, 1994.

[18] K. Svozil, *Randomness and Undecidability in Physics,* World Scientific, Singapore, 1993, pp. 30-35.